Locally Static, Globally Dynamic Session Types for Active Objects

Reiner Hähnle
Technical University Darmstadt, Germany
reiner.haehnle@tu-darmstadt.de

Anton W. Haubner
Technical University Darmstadt, Germany
anton.haubner@stud.tu-darmstadt.de

Eduard Kamburjan
Technical University Darmstadt, Germany
kamburjan@cs.tu-darmstadt.de

Abstract

Active object languages offer an attractive trade-off between low-level, preemptive concurrency and fully distributed actors: syntactically identifiable atomic code segments and asynchronous calls are the basis of cooperative concurrency, still permitting interleaving, but nevertheless being mechanically analyzable. The challenge is to reconcile local static analysis of atomic segments with the global scheduling constraints it depends on. Here, we propose an approximate, hybrid approach; At compile-time we perform a local static analysis: later, any run not complying to a global specification is excluded via runtime checks. That specification is expressed in a type-theoretic language inspired by session types. The approach reverses the usual (first global, then local) order of analysis and, thereby, supports analysis of open distributed systems.

2012 ACM Subject Classification Theory of computation → Distributed computing models; Theory of computation → Object oriented constructs; Theory of computation → Type structures

Keywords and phrases Session Types, Active Objects, Runtime Verification, Static Verification

Digital Object Identifier 10.4230/OASIcs.Gabbrielli.2020.1

Funding This research is supported by the *Constraint-Based Operational Consistency of Evolving Software Systems* (`COCoS`) project, funded by the DFG as project 351097374.

For Maurizio Gabbrielli:
"Les raisins, ou la mort!"

1 Introduction

Lately, programming languages based on actors and *active objects* (AO) attracted a lot of interest in both academia and industry. Active objects [11] are an object-oriented modeling formalism, extending the actor model of distributed systems [21]. One prominent representative, the *abstract behavioral specification* (`ABS`) [24] language, was successfully applied in a variety of domains, ranging from railway operations [31] to cloud-based systems [40].

One of the advantages of `ABS` is its rich analysis framework with tools based on dataflow and graph analyses [3], deductive verification [13], and behavioral types [18]. However, for the time being, there is no support for code generation from scheduling policies (except user-defined schedulers at the object level), or for runtime verification beyond simple **assert** statements. The reasons lie in the AO (ABS) concurrency model.

Communication between Active Objects. An Active Object is a strongly encapsulated entity whose fields can only be accessed by getter and setter methods. Like an object in standard OO, an AO declares a set of methods, including constructors. Its peculiarity is that each method consists of *syntactically* marked atomic segments whose execution cannot be preempted. At most one task, executing the code of an atomic segment, is active at any time on the object's single processor. The advantage is that each atomic segment functions as a sequential program and can be analyzed (method-)*locally* in a *modular* fashion. However, its scheduling condition may depend on the availability of results provided by other methods, not necessarily from the same object. But such synchronization patterns can only be understood from an *object*-local or even *global* (program-wide) perspective. This is bad news for the local analysis of atomic segments as well, because in general they require information about previously scheduled tasks in order to guarantee meaningful properties. This dependency is an obstacle to any modular, global analysis. While it is possible to write sufficiently strong local contracts [30], it is a difficult, manual task, which does not align well with a top-down design starting from global communication patterns. In consequence, the ability to verify closed systems (that do not interact with their environment) is limited. Even worse, the analysis of open `ABS` models is generally impossible.

Additionally to cooperative method contracts [30], `ABS` currently uses Session Types [28, 29] for the verification of communication patterns. Both approaches consist of a *local* part that analyzes the code of single methods and a *global* part, feeding into it, that analyzes scheduling, synchronization, and messages. The global part is significantly more imprecise than the local one, because it abstracts away from functional behavior. The local specification is related to the global specification via a process called projection (from objects down to methods and atomic segments): the composition principle of the analysis follows the composition principle of the AO concurrency model [19].

Locally Static, Globally Dynamic Approach. In this paper we reverse the analysis sequence and partially move it from compile-time to runtime, resulting in a *hybrid* verification method. Specifically, local analysis is done statically, at compile-time, while global analysis is performed *later* at runtime. This is achieved by a modification of the workflow of session types: Classically, projection ensures that messages always arrive in their correct order. We retain projection, but only infer the *correct message order per object*, then construct a scheduler that enforces this order.

Local static checks permit to verify open systems: a locally specified `ABS` model provides only methods that perform locally correct steps, while at runtime it is ensured that methods are called correctly and in correct order. The downside is, obviously, that global errors are only detected at runtime, however, in an open system this is the only option. The second limitation of our approach is that it is not designed to perform full functional verification of state invariants, unlike interactive, deductive verification [30]. We aim at a lightweight, fully automatic method that nevertheless allows to express non-trivial properties and facilitates top-down design of distributed systems.

Yet, our approach does not modify the `ABS` concurrency model and requires as the single extension the availability of user-defined schedulers, i.e., the ability to reject certain task sequences. From the point of modularity, we can now verify *object*-local behavior. We implemented and evaluated our approach and illustrate with a case study that it is possible to ensure an open system always follows a given protocol.

Structure. In Sect. 2 we introduce active objects, `ABS`, and a suitable notion of session types. Sect. 3 describes scheduler generation and instrumentation, Sect. 4 describes and evaluates the implementation. In Sect. 5 we discuss related approaches. Finally, Sect. 6 concludes and gives future work. For space reasons, here we can describe the main ideas only with a limited degree of precision. A fully formal treatment is found in the report [20].

2 Active Objects and Session Types

The concurrency model of AO, as explained above, rests on a syntactically identifiable notion of atomic code segments that cannot be preempted. Together with strong encapsulation, this ensures that an object's state can only be modified by its own methods (including setters) and any state change must adhere to the local specification of an atomic segment. This is the basis to establish an object's invariant by suitable, *cooperative scheduling* of methods and their atomic segments. To make this work it is necessary to call a method of another (or even the own) object *without blocking*.

All active object languages, therefore, feature non-blocking, *asynchronous* method calls that return a future [4], a handle to the task executing the call.

$$\begin{array}{rlr}
\mathsf{Prgm} ::= & \overrightarrow{\mathsf{ID}}\ \overrightarrow{\mathsf{CD}}\ \mathsf{Main} \qquad \mathsf{ID} ::= \ \textbf{interface}\ \texttt{I}\ \left[\textbf{extends}\ \overrightarrow{\texttt{I}}\right]?\{\overrightarrow{\mathsf{MS}}\} & \text{Program, Interfaces}\\
\mathsf{CD} ::= & \textbf{class}\ \texttt{C}\ \left[\textbf{implements}\ \overrightarrow{\texttt{I}}\right]?\ \left[(\overrightarrow{\texttt{T f}})\right]?\{\overrightarrow{\mathsf{FD}}\ \overrightarrow{\mathsf{Met}}\ \mathsf{Run}\} \qquad \mathsf{Main} ::= \ \{\mathsf{s}\} & \text{Classes, Main}\\
\mathsf{Run} ::= & \texttt{Unit run()}\ \{\mathsf{s}\} \qquad \mathsf{FD} ::= \ \texttt{T f = e} & \text{Run Method, Fields}\\
\mathsf{MS} ::= & \texttt{T m(}\overrightarrow{\texttt{T v}}\texttt{)} \qquad \mathsf{Met} ::= \ \mathsf{MS}\ \{\mathsf{s};\ \textbf{return}\ \texttt{e};\} & \text{Signatures, Methods}\\
\mathsf{s} ::= & \textbf{while}\ \texttt{(e)}\ \{\mathsf{s}\} \mid \textbf{if}\ \texttt{(e)}\ \{\mathsf{s}\}\ [\textbf{else}\ \{\mathsf{s}\}]? \mid \mathsf{s};\ \mathsf{s} & \\
 & \mid \textbf{case}\ \texttt{(e)}\ \{\overrightarrow{\texttt{e => s;}}\} \mid \textbf{await}\ \mathsf{g} \mid [\texttt{T}?\ \texttt{e}]? = \mathsf{rhs} & \text{Statements}\\
\mathsf{g} ::= & \texttt{e?} \qquad \mathsf{rhs} ::= \ \texttt{e} \mid \textbf{new}\ \texttt{C(}\overrightarrow{\texttt{e}}\texttt{)} \mid \texttt{e.}\textbf{get} \mid \texttt{e!m(}\overrightarrow{\texttt{e}}\texttt{)} & \text{Guards and RHS's}
\end{array}$$

Figure 1 `ABS` grammar. `T` ranges over types, `I` over interfaces and `C` over classes.

The various AO languages differ in the details of how synchronization is performed, so we now turn to their specific realization in `ABS`. The syntax of `ABS` is given by the grammar in Fig. 1. With `e` we denote standard expressions over fields `f`, variables `v` and operators `|`, `&`, `>=`, `<=`, `+`, `-`, `*`, `/`. Additionally, we use an expression `destiny` to access the currently computed future. Types `T` are all interface names (`ABS` enforces programming to interfaces), type-generic futures **Fut**`<T>`, lists `List<T>`, `Int`, `Unit`, and `Bool`. We also assume the usual functions for lists, etc.

In the final expression of the rule for rhs, the syntax for asynchronous method calls is shown (for simplicity, we leave out standard synchronous calls). As usual, a "`!`" replaces the dot. Asynchronous calls are always executable. Their result is a *future* of type **Fut**`<T>`, where `T` is the return type of `m`. The *effect* of an asynchronous call is to create a task to execute `m`'s body in `e`'s object `o`, to be scheduled at some time in the future. In case `o` is also the caller, obviously the calling method must first suspend, before the callee can be scheduled. Asynchronous calls occur only as right-hand side expressions, so the future is stored in a field (or variable) `f`. Once the result of the computation performed by $\texttt{m(}\overrightarrow{\texttt{e}}\texttt{)}$ is ready, it can be retrieved with the expression `f.`**get**. If the result is *not* ready, the **get** expression blocks the calling object. This can easily lead to a deadlock, so one typically *guards* a **get** expression with an **await** statement of the form **await** `e?`, where `e`'s type is of the form **Fut**`<T>`. The effect is that execution of the current task is suspended and only rescheduled after the result of `e` is ready. The **await** statement and the syntactic end of a method block *are the only places, where task suspension in* `ABS` *can occur.* This justifies the following definition:

▶ **Definition 1** (Atomic Segment). *Code sequences starting either at the syntactic beginning of a method body or at the statement right after an* **await** *statement and ending either at the syntactic end of a method body or with an* **await** *statement, such that they contain at most the* **await** *statement at the end, are called* atomic segment.

Generally, ABS programs follow a simple, but standard OO paradigm in any other aspect: A program contains a main method Main, interfaces $\overrightarrow{\mathsf{ID}}$ and classes $\overrightarrow{\mathsf{CD}}$. Interfaces are standard, the main method contains a list of object creations. Classes can have parameters $\overrightarrow{\mathsf{Tf}}$, these are fields being initialized during object creation. The parameter type may have the form **Fut**<T>, i.e., futures may be passed as arguments to other methods. Classes have fields $\overrightarrow{\mathsf{FD}}$, methods $\overrightarrow{\mathsf{Met}}$, and a run method Run to start a process.

▶ **Example 2.** Let us illustrate cooperative scheduling in ABS with the example in Fig. 2. The program models the behavior of a mail server with notification service. It consists of three objects created in the main block: a mail server m, a user interface u, and a notification service n, which knows both the mail server and the user interface. Then the notification service's only method init() is run on n. The method has a single loop that periodically checks whether mail arrived and, if this is the case, notifies the user via interface u. Checking for mail and notifying the user require asynchronous calls to m and u, respectively, so we allocate suitable fields fCheckMail and fPopup of future type. Checking the mail must be finished before notification is handled. This is ensured by the **await** statement in Line 12. At this point, init() suspends. In the example, init() is the only method executing on m, so the processor will be simply idle, but it is conceivable that the main method starts other tasks on the mail server which at this point can be interleaved. Checking for mail is modelled by randomly choosing one of the literals Mail or NoMail as a return value.

Once the response is available, the user is notified in case there is mail, otherwise, nothing happens. Since the call to popup() is asynchronous, in the absence of a defined scheduler, the sequence of multiple calls to popup() is not necessarily in the order of mail arriving. However, the code ensures that the number of completed or active calls to popup() is always less than the completed calls to checkMail(), that there can be at most one call to popup() between any two calls to checkMail(), etc. We will show that session types are suitable to specify such global behavior in a succinct way, which then can be enforced at runtime.

Before we define session types for AO, we need to set up the machinery of user-defined schedulers needed to implement runtime checks.

A user-defined scheduler [5] is a side-effect free function in ABS that takes as parameters (1) a list of schedulable *processes* and (2) several fields of its class. It returns either Nothing or Just(p), where p is one of the processes in the input list. The return value controls scheduling: if Nothing is returned, no process is scheduled, otherwise the chosen process is scheduled next. A process is represented as an abstract data type Process, i.e., an ADT that cannot be constructed manually. Instead one can access the future of a process with destinyOf(p) and its methodname as a String with method(p).

Lists are also ADTs and nth(input,i) returns the i-th element. Keyword **def** is used to define a function with parameters that evaluates a result using standard expressions (and recursion). ABS does not support fully-fledged functional programming and only a fixed set of higher-order functions. For example, higher-order functions such as map and filter are part of the ABS standard library.

▶ **Example 3.** Let us consider the following scheduler and class.

```
def Maybe<Process> scheduler(List<Process> input, Int y, String m) =
  if ( y < 0 || y >= length(input) ) Nothing else
    if ( method(nth(input,y)) != m) Just(nth(input,y) ) else Nothing;

[Scheduler: scheduler(queue, y, m)]
class C(String m, Int y) { ... }
```

```
data Msg = Mail | NoMail;

class NotifyService
      (MailServerI m, UII u)
      implements NotifyServiceI {
 Fut<Msg>  fCheckMail;
 Fut<Unit> fPopup;
 Unit init(int bound) {
  Int i = 0;
  while (i < bound) {
   fCheckMail = m!checkMail();
   await fCheckMail?;
   Msg response = fCheckMail.get;
   case (response) {
    Mail => fPopup = u!popup(i);
    NoMail => skip;
   }
   i = i + 1;
  }
 }
}
class MailServer
      implements MailServerI {
 Msg checkMail() {
  Msg result = NoMail;
  if (random(2) == 1)
      result = Mail;
  return result;
 }
}
class UI implements UII {
 Unit popup(int id) {
  println("You got mail! Id " + id);
 }
}
{ // Main block
 MailServerI m = new MailServer() ;
 UII u = new UI();
 NotifyServiceI n
    = new NotifyService(m,u);
 await n!init(42);
}
```

Figure 2 Mail server example in ABS.

y and m, the field names and their types, must be identical in class and scheduler function to use the scheduler. The code above selects the y-th element in the input list, unless it is out of range or a process executing a method named m. The annotation `[Scheduler: scheduler(queue, y, m)]` connects scheduler and class. Whenever the scheduler is invoked, the input list is guaranteed to be non-empty.

2.1 Session Types for Active Objects

Session types specify and verify the behavior of a closed unit of communication, called a *session*. A session type specification consists of three parts: (1) global types, a global specification of the session, (2) local types, specifications for the endpoints in a session, and (3) a projection mechanism that generates a local specification for each endpoint participating in the communication from a global type. For checking that the whole unit adheres to its global specification, it suffices to check the local endpoints and, possibly, side-conditions on the unit. Additionally, the session type system needs some kind of mechanism to ensure that the local endpoints adhere to their local type.

In the original formulation for the π-calculus [8] a session is centered around a channel, endpoints are the processes participating in the communication over the typed channel. To ensure that projection succeeds, a linearity check on the channel is performed as a side-condition of projection.

For Active Objects, the situation changes: there are no channels and endpoints participating in any communication are not uniform, because the target of a method call is an object, but the target of a future read is a (terminated) process. As there are no channels, a linearity check to ensure that messages arrive in the right order is impossible.

Session types for AO [28, 29] adopt and adapt the concepts of session types for channels:

Unit of Communication: The unit of communication is described by a set of objects that only contain pointers to each other.

Endpoints: The notion of endpoint is two-fold. Both objects and processes are endpoints and the projection of a global session type first projects on the object and then projects one more time on the processes inside that object. The result of the first projection is an *object-local* type and the result of the second projection is a *method-local* type.[1]

Order of Messages: To ensure messages arrive in the correct order, a static analysis can be used to determine whether the order of messages is total from the perspective of each object (but not globally).

Here we remove the check on message order at the level of the type system and instead enforce it at runtime using the structure provided by the object-local type. Before we introduce syntax of global and local types, it is worth mentioning that we are only concerned with *protocol adherence*: Does the system implement the protocol described by the global type? We ignore *deadlock freedom*, which can be approached either with session types [29] or a dedicated deadlock checker for AO [15, 18, 25]. The system we introducing below is a slight variation of the session types in [29].

2.1.1 Global Types

Global types follow the structure of regular expressions and allow Kleene star-style repetition, sequence and branching. Branching is guarded by a single role that determines which branch of the protocol to follow. As single actions, the type defines a certain kind of interaction between two roles or a role and a process/future. To keep track of processes and futures within a protocol, we use tracked futures: references to the future of a specified method call.

▶ **Definition 4.** *Let* $\mathbf{p}$, $\mathbf{q}$ *range over roles,* t *over tracked futures and* C *over ADT constructors. The syntax of global protocols* $\mathbf{GP}$ *and global types* $\mathbf{G}$ *is defined in Fig. 3. Specifications* $(\!|\cdot|\!)$ *are all optional.*

$$\begin{array}{rll}
\mathbf{GP} ::= & \mathbf{0}\xrightarrow{\mathsf{t}}\mathbf{p}{:}\mathtt{m}\ .\ \mathbf{G} & \text{Global Protocol} \\
\mathbf{G} ::= & \mathbf{p}\xrightarrow{\mathsf{t}}\mathbf{q}{:}\mathtt{m} \mid \mathbf{p}{\downarrow}\mathsf{t}(\!|\mathsf{C}|\!) \mid \mathbf{p}{\uparrow}\mathsf{t}(\!|\mathsf{C}|\!) & \text{Call, Termination and Synchronization Action} \\
 & \mid \mathsf{Rel}(\mathbf{p},\mathsf{t}) \mid \mathbf{skip} & \text{Suspension and Empty Action} \\
 & \mid \mathbf{p}\{\mathbf{G}_i\}_{i\in I} \mid (\mathbf{G})^{*} \mid \mathbf{G}\ .\ \mathbf{G} & \text{Branching, Repetition and Sequential Composition}
\end{array}$$

Figure 3 Syntax of Global Session Types.

The global protocol starts with $\mathbf{0}\xrightarrow{\mathsf{t}}\mathbf{p}{:}\mathtt{m}$ and specifies how the session is started. The call action $\mathbf{p}\xrightarrow{\mathsf{t}}\mathbf{q}{:}\mathtt{m}$ specifies a call from the object with role $\mathbf{p}$ to the one with role $\mathbf{q}$ on method $\mathtt{m}$. This process is tracked by t in the rest of the type. The termination action $\mathbf{p}{\downarrow}\mathsf{t}(\!|\mathsf{C}|\!)$ specifies that the object with role $\mathbf{p}$ terminates the process tracked by t and the return value has the outermost constructor C. The synchronization action $\mathbf{p}{\uparrow}\mathsf{t}(\!|\mathsf{C}|\!)$ specifies that the object with role $\mathbf{p}$ reads from the future tracked by t and reads a value with the outermost constructor C. The suspension action $\mathsf{Rel}(\mathbf{p},\mathsf{t})$ specifies that the object with role $\mathbf{p}$ suspends its currently active process until the future tracked by t is resolved. We stress

[1] The projection may also be done in one step [27], but this removes the object-local types which we are investigating in this paper.

that $\mathtt{t}$ is *not* the tracked future of the suspended process. The empty action specifies no action and is needed to specify, e.g., branches without visible actions. Branching $\mathbf{p}\{\mathbf{G}_i\}_{i\in I}$ specifies that the object with role $\mathbf{p}$ chooses one of the branches $\mathbf{G}_i$ to continue the protocol. Finally, repetition and sequential composition are analogous to regular expressions.

▶ **Example 5.** We continue Ex. 2. The roles of the protocol are named **NS** for the notification service, **Mail** for the mail server and **GUI** for the GUI. The intended behavior is specified by the following global type:

$$
\begin{array}{l}
\mathbf{0} \xrightarrow{\mathtt{t}_0} \mathbf{NS}:\mathtt{init}\ .\\
\big(\mathbf{NS} \xrightarrow{\mathtt{t}_1} \mathbf{Mail}:\mathtt{check}\ .\ \mathsf{Rel}(\mathbf{NS},\mathtt{t}_1)\ .\\
\mathbf{Mail}\left\{\begin{array}{l}\mathbf{Mail}\!\downarrow\!\mathtt{t}_1(\!|\mathtt{NewMail}|\!)\ .\ \mathbf{NS}\!\uparrow\!\mathtt{t}_1(\!|\mathtt{NewMail}|\!)\ .\ \mathbf{NS} \xrightarrow{\mathtt{t}_2} \mathbf{GUI}:\mathtt{show}\ .\ \mathbf{GUI}\!\downarrow\!\mathtt{t}_2\\ \mathbf{Mail}\!\downarrow\!\mathtt{t}_1(\!|\mathtt{NoMail}|\!)\ .\ \mathbf{NS}\!\uparrow\!\mathtt{t}_1(\!|\mathtt{NoMail}|\!)\end{array}\right\}\\
\big)^*\ .\ \mathbf{NS}\!\downarrow\!\mathtt{t}_0
\end{array}
$$

The above example demonstrates the use of tracked futures and repetition, but it is strongly synchronized: The described synchronization structure enforces correct interaction order, no deviation due to the scheduler is possible. In contrast, consider the following scenario and global session type that is not strongly synchronized.

▶ **Example 6.** The protocol describes four roles: a student **S**, a service desk **D**, a computation server **C** and a report generator **R**. The computation server computes the grade of a student, and sends it to the report generator, which in turn generates a report that is send to the service desk. The computation server notifies the student that its grade has been computed. The service desk may only serve the student after the report has arrived. This is specified by the following global protocol:

$$
\begin{array}{l}
\mathbf{0} \xrightarrow{\mathtt{t}_0} \mathbf{C}:\mathtt{compute}\ .\ \mathbf{C} \xrightarrow{\mathtt{t}_1} \mathbf{R}:\mathtt{toReport}\ .\ \mathbf{C} \xrightarrow{\mathtt{t}_2} \mathbf{S}:\mathtt{notify}\ .\ \mathbf{R} \xrightarrow{\mathtt{t}_3} \mathbf{D}:\mathtt{publish}\ .\ \mathbf{R}\!\downarrow\!\mathtt{t}_1\\
.\ \mathbf{D}\!\downarrow\!\mathtt{t}_3\ .\ \mathbf{S} \xrightarrow{\mathtt{t}_4} \mathbf{D}:\mathtt{request}\ .\ \mathbf{D}\!\downarrow\!\mathtt{t}_4\ .\ \mathbf{S}\!\uparrow\!\mathtt{t}_4\ .\ \mathbf{S}\!\downarrow\!\mathtt{t}_2\ .\ \mathbf{C}\!\downarrow\!\mathtt{t}_0
\end{array}
$$

Note that **D** is called on `request` and `publish` but no synchronization ensures that those messages arrive in the specified order.

2.1.2 Local Types

We distinguish object-local types, method-local types and scheduling types. Method-local and object-local types differ syntactically only in their passive choice operator and the specification of synchronization. Scheduling types describe the actions of the scheduler of a role and share their syntax with method-local types.

▶ **Definition 7.** *Let* $\mathbf{p}$ *range over roles and* $\mathbf{0}$, $\mathtt{m}$ *over method names,* $\mathtt{t}$ *over tracked futures and* C *over ADT constructors. The syntax of object-local types* $\mathbf{L}$ *is defined as follows:*

$\mathbf{L} ::= \mathbf{p}?_{\mathtt{t}}\mathtt{m} \mid \mathbf{p}!_{\mathtt{t}}\mathtt{m} \mid \mathsf{Put}\ \mathtt{t}(\!|\mathsf{C}|\!)$	Receiving, Sending and Termination Action
$\mathsf{Get}\ \mathtt{t}(\!|\mathsf{C}|\!) \mid \mathsf{Susp}(\mathtt{t},\mathtt{t}) \mid \mathsf{React}\ \mathtt{t}$	Synchronization, Suspension and Reactivation Action
$\&_{\mathtt{t}}\{\mathbf{L}_i\}_{i\in I} \mid \oplus\{\mathbf{L}_i\}_{i\in I}$	Passive and Active Choice
$\mathbf{skip} \mid \mathbf{L}\ .\ \mathbf{L} \mid (\mathbf{L})^*$	Empty Action, Sequential Composition and Repetition

The syntax of method-local types is analogous, but (1) the synchronization action takes no C *specification and (2) passive choice takes the following form, called* guarded *passive choice:*

$$\&_{\mathtt{t}}\{\mathsf{C}_i : \mathbf{L}_i\}_{i\in I}$$

The receiving action is the callee's view on the global call action, $\mathbf{p}$ is the caller. Note that a method in ABS has no access to the caller, but we may access it in the scheduler. The sending action is the caller's view on the global call action, $\mathbf{p}$ is the callee. The local termination action is the equivalent of the global termination action. The local suspension action $\mathsf{Susp}(\mathtt{t}_1, \mathtt{t}_2)$ specifies that the process computing $\mathtt{t}_1$ suspends until $\mathtt{t}_2$ is known. The reactivation action $\mathsf{React}\ \mathtt{t}$ specifies reactivation of the process computing $\mathtt{t}$. These two actions are the local view on the global suspension action, but (1) locally the suspending process is known and (2) one can infer, where the reactivation must happen. We use three choice operators:

- (Object-Local) Unguarded passive choice $\&_{\mathtt{t}}\{\mathbf{L}_i\}_{i\in I}$ specifies that the object reacts to the choice stored in $\mathtt{t}$. The choice is stored as the C parameter of the first Get action on $\mathtt{t}$ in the given branch.
- (Method-Local) Guarded passive choice $\&_{\mathtt{t}}\{\mathsf{C}_i : \mathbf{L}_i\}_{i\in I}$ specifies which constructor corresponds to which branch directly, as it is only indirectly encoded in the unguarded passive choice.
- Active choice $\oplus\{\mathbf{L}_i\}_{i\in I}$ specifies that the object or process in question chooses one of the branches to continue. It is not specified how the choice is made.[2]

The remaining actions are analogous to their global counterpart.

We introduce projection formally in the subsequent section, but provide examples of local types based on Ex. 5, 6 now to illustrate the differences among the various local types.

▶ **Example 8.** Below is the object-local type of **Mail** in Ex. 5 followed by the method-local type of $\mathtt{t}_1$ and the scheduling type. The differences between the former are that (1) the method-local type contains *no* repetition, because the repetition is not visible to a single process and (2) the receiving action is omitted, because it is redundant when the tracked future is known.

$$\begin{array}{ll}
\Big(\mathbf{NS}?_{\mathtt{t}_1}\mathtt{check}\ .\ \oplus\left\{\begin{array}{l}\mathsf{Put}\ \mathtt{t}_1\,(\!|\mathtt{NewMail}|\!)\\ \mathsf{Put}\ \mathtt{t}_1\,(\!|\mathtt{NoMail}|\!)\end{array}\right\}\Big)^* & \text{Object-local type}\\
\qquad\qquad\qquad\ \ \oplus\left\{\begin{array}{l}\mathsf{Put}\ \mathtt{t}_1\,(\!|\mathtt{NewMail}|\!)\\ \mathsf{Put}\ \mathtt{t}_1\,(\!|\mathtt{NoMail}|\!)\end{array}\right\} & \text{Method-local type}\\
\Big(\mathbf{NS}?_{\mathtt{t}_1}\mathtt{check}\qquad\qquad\qquad\qquad\Big)^* & \text{Scheduling type}
\end{array}$$

The method-local type contains only the actions performed by the processes of a single method. A scheduling type contains only the actions needed for scheduling: empty, reactivation and receiving actions, as well as both kinds of branching, repetition and sequential compositions. The following is the scheduling type of **D** in Ex. 6: $\mathbf{R}?_{\mathtt{t}_3}\mathtt{publish}\ .\ \mathbf{S}?_{\mathtt{t}_4}\mathtt{request}$

3 LSGD Session Types

The verification workflow of our system takes a global type and generates an instrumented ABS program and a proof that each method is following its method-local type.

- First, we establish certain well-formedness conditions of the global type to ensure it describes a protocol that is realizable in the AO concurrency model.

[2] For guarded active choice we refer to Kamburjan & Chen [28].

- Then, the global type is projected on each participating object. This results in an object-local type, describing the actions an object both expects and is obliged to perform.
- From the object-local type we generate (a) a session automaton that describes the order of scheduling actions and (b) a method-local type for each method. Scheduling actions include the receiving action (receiving method calls) and the rescheduling action (reacting to a resolved future).
- The session automaton is translated into a user-defined scheduler, which is added to the object together with fields and operations to keep track of the state.
- Each method is checked statically against its method-local type.

For brevity, we give a simplified account of the implemented system [20] and omit some features, e.g., allowing interactions with objects that do not participate in the session.

3.1 Session Automata

Before defining the workflow, we introduce Session Automata [7]. Session automata are a class of register automata [33]: finite automata over an infinite alphabet. General register automata allow to store read values of infinite alphabets in registers and compare the register contents by equality. Session automata have the restriction that only *fresh values* can be stored, i.e., values that have not been seen in the input word so far. This matches our model when futures are regarded as data and allows one to decide whether two session automata accept the same language. In our system, the alphabet is the set of futures and we only store futures upon receiving a method call. This guarantees their freshness upon storage.

▶ **Definition 9.** *Let Σ be a finite set of labels, D an infinite set of data equipped with equality and $k \in \mathbb{N}$. A k-Register Session Automaton is a tuple (Q, q_0, Φ, F), where Q is the set of states, $q_0 \in Q$ its start state, $F \subseteq Q$ the set of accepting states, and the transition relation is as follows:*

$$\Phi \subseteq \big(Q \times Q\big) \cup \big(Q \times (\Sigma \times D) \times \mathcal{P}(\{1, \dots, k\}) \times \{1, \dots, k\} \times Q\big)$$

Runs of session automata are defined over stores and data words. A transition either (1) only changes the state, but neither changes the store nor consumes a letter, or (2) changes the state upon reading the next letter by comparing the data with a register in its store and storing the read data.

▶ **Definition 10.** *A store $\sigma : \{1, \dots, k\} \mapsto D \cup \{\bot\}$ is a function from register identifiers to data or the special symbol $\bot$. The initial store σ_0 maps all register identifiers to $\bot$. A data word $w = (a_0, d_0), \dots, (a_n, d_n)$ is a finite sequence of pairs of labels and data. A run $(q_0, j_0, \sigma_0), \dots, (q_m, j_m, \sigma_j)$ of a k-register session automaton (Q, q_0, Φ, F) on a word w of length n is a sequence*

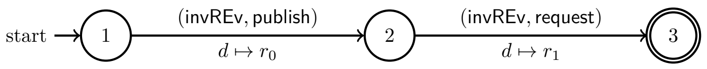
$$s \in (Q \times \mathbb{N} \times \{1, \dots, k\} \mapsto D)^*$$

where q_i is the current state, σ_i the current store and j_i the next letter. The sequence must start with $(q_0, 0, \sigma_0)$ and satisfy the following condition for each position $0 < i < m$:

$$\begin{aligned} &(q_i, q_{i+1}) \in \Phi \wedge (j_i = j_{i+1}) \wedge (\sigma_i = \sigma_{i+1}) \\ \vee &\Big((q_i, (a_{j_i}, d_{j_i}), I, k, q_{i+1}) \in \Phi \wedge (j_i = j_i + 1) \wedge \sigma_{i+1} = \sigma_i[k \setminus d_{j_i}] \wedge \forall l \in I.\ \sigma_i(l) = d_{j_i}\Big) \end{aligned}$$

In the following we set $D = \mathsf{Fut}$ and $\Sigma = \{\mathsf{invREv}\} \times \mathsf{Met} \cup \{\mathsf{condREv}\}$.

▶ **Example 11.** The following 2-register session automaton models the scheduling type of $\mathbf{D}$ in Ex. 8. The two stores of the futures in registers r_i are used to model reactivation.



For brevity, we write $(q, (\mathsf{invREv}, \mathtt{m}), q')$ and $(i, (\mathsf{condREv}), q')$ for transitions with the given label and say that register i is either written or read. We never write to or read from more than one register in a single transition.

3.2 Projection

Projection generates (1) a method-local type per participating method in the session and (2) a special object-local type, called *scheduling type*, for each role. The scheduling type describes the order of operations controlled by the scheduler, i.e., process start and rescheduling.

Projection consists of four steps: pre-analysis, projection on a role, projection on a tracked future, and generation of a scheduling type from an object-local type.

Pre-analysis: Reject obviously malformed types and annotate the global type with information used in later steps, for example, which future is currently being computed.

Projection on Role: Generate an object-local type that describes the view of a role on the global type.

Projection on Tracked Future: Generate a method-local type that describes the view of a process on the object-local type.

Generation of Scheduling Type: Generate the scheduling type that describes the operations performed by the scheduler of an object.

3.2.1 Pre-Analysis

Pre-analysis of a global type checks that it specifies a feasible protocol in the AO concurrency model. It generates an *annotated* global type $\mathbf{G}\langle\sigma\rangle$, where σ describes the specified state of a role before and after performing the specified action. We refrain from introducing all the formal details and only describe the checked properties of a global type.

Future Freshness: Each tracked future identifies exactly one call action. For example, the following type fails pre-analysis and is rejected, because t is not fresh in the second call.

$$\mathbf{0} \xrightarrow{\mathsf{t}} \mathbf{p}{:}\mathtt{m} \;.\; \mathbf{p} \xrightarrow{\mathsf{t}} \mathbf{q}{:}\mathtt{n} \;.\; \mathbf{p}{\downarrow}\mathsf{t} \;.\; \mathbf{q}{\downarrow}\mathsf{t}$$

Actor Activity: A call action can only be specified when the callee is not specified as currently executing a method and a suspending action can only suspend a process when it is specified as being active. For example, the following global type contains two errors: the call of t_2 must wait until $\mathbf{p}$ is terminated and the suspension action of $\mathbf{p}$ cannot suspend any process.

$$\mathbf{0} \xrightarrow{\mathsf{t}_0} \mathbf{p}{:}\mathtt{m} \;.\; \mathbf{p} \xrightarrow{\mathsf{t}_1} \mathbf{q}{:}\mathtt{n} \;.\; \mathbf{q} \xrightarrow{\mathsf{t}_2} \mathbf{p}{:}\mathtt{o} \;.\; \mathbf{p}{\downarrow}\mathsf{t}_0 \;.\; \mathsf{Rel}(\mathbf{p}, \mathsf{t}_1) \;.\; \mathbf{q}{\downarrow}\mathsf{t}_1 \;.\; \mathbf{p}{\downarrow}\mathsf{t}_2$$

The following is one possible “debugged” version that passes pre-analysis:

$$\mathbf{0} \xrightarrow{\mathsf{t}_0} \mathbf{p}{:}\mathtt{m} \;.\; \mathbf{p} \xrightarrow{\mathsf{t}_1} \mathbf{q}{:}\mathtt{n} \;.\; \mathsf{Rel}(\mathbf{p}, \mathsf{t}_1) \;.\; \mathbf{q} \xrightarrow{\mathsf{t}_2} \mathbf{p}{:}\mathtt{o} \;.\; \mathbf{p}{\downarrow}\mathsf{t}_2 \;.\; \mathbf{q}{\downarrow}\mathsf{t}_1 \;.\; \mathbf{p}{\downarrow}\mathsf{t}_0$$

Resolution Analysis: A future can only be read if it has terminated before and is accessible to the reading role.[3]

Scope Analysis: Repetition introduces *scopes* into the specification, as a process is started exactly once and terminated exactly once. For example, the following type is not correctly scoped, because it allows situations where (1) $\mathtt{n}$ is never called, and thus $\mathtt{t}_1$ cannot be terminated and (2) where $\mathtt{n}$ is called multiple times and it is not specified how many of those processes are terminated and in which order:

$$\mathbf{0} \xrightarrow{\mathtt{t}_0} \mathbf{p}:\mathtt{m}\ .\ (\mathbf{p} \xrightarrow{\mathtt{t}_1} \mathbf{q}:\mathtt{n})^*\ .\ \mathbf{p}\!\downarrow\!\mathtt{t}_1\ .\ \mathbf{q}\!\downarrow\!\mathtt{t}_0$$

The scope analysis checks that (1) every tracked future that is started within a repetition is resolved within the same repetition; (2) every tracked future that is resolved within a repetition is started within the same repetition; (3) every tracked future that is synchronized upon within a repetition is started within the same repetition; (4) for every role the active tracked future and the set of suspended tracked future before and after the repetition are the same. (5) every tracked future that is started within a branch is resolved within the same branch; (6) every tracked future that is resolved within a branch is started within the same branch;

During pre-analysis each global type, except sequence, is annotated with an abstract state σ. An abstract state is a mapping from roles to a pair $(\mathit{AState}, \mathit{SState})$, where AState is either $\mathsf{Active}(\mathtt{t})$, expressing that the role is currently specified as executing the process for $\mathtt{t}$ or Susp if it is currently specified inactive. SState is a set of pairs of tracked futures $(\mathtt{t}, \mathtt{t}')$, expressing that there is a suspended process for $\mathtt{t}$ waiting for $\mathtt{t}'$. Pre-analysis ensures that there are no $\mathtt{t}_1, \mathtt{t}_2$ with $(\mathtt{t}_1, \mathtt{t}), (\mathtt{t}_2, \mathtt{t}) \in \mathit{SState}$ in any abstract state for any role, i.e., there are never two processes of one role waiting for the same future.[4]

3.2.2 Global Projection

The projection of global types on a role is defined in Fig. 4. Projection is a partial function $\mathbf{G} \upharpoonright \mathbf{p}$. It checks that any action is specified to happen when the role performing this action is active and has a process that can perform the communication. The result of projection is an object-local type, annotated with abstract states.

The initial action results in a receiving action for the callee and **skip** for any other role. Similarly, the projection of a call action is a receiving action for the callee and a sending action for the caller.

Projection of the termination action has three cases: (1) If projected on the terminating role, it is ensured that this role is active and can perform the action. The result is a local termination action. (2) If projected on a role waiting for the tracked future of the action, it is ensured that this role is inactive. The result is a reactivation action. (3) Projection on any other role results in **skip**. Projection fails if, for example, the terminating role is inactive.

Projection of synchronization results in a local synchronization action for the specified role and **skip** for any other role. It is checked that the specified role is active. The suspension action is analogous. Projection of **skip** is the identity, projection of branching results in an active choice for the specified role (which must be active) and a passive choice over the currently active future of the choosing role for any other role. Projection of the repetition

[3] On passing data in Session Types for Active Objects, we refer to [27].

[4] Because it is not *specified* in which order they should be reactivated. If such a specification were given, that order could be reflected in the projected object-local type.

$$\mathbf{0} \xrightarrow{\mathsf{t}} \mathbf{q} : \mathtt{m}\langle\sigma\rangle \restriction \mathbf{p} = \begin{cases} \mathbf{0}?_{\mathsf{t}}\mathtt{m}\langle\sigma\rangle & \text{if } \mathbf{p} = \mathbf{q} \\ \mathbf{skip} & \text{otherwise} \end{cases}$$

$$\mathbf{q} \xrightarrow{\mathsf{t}} \mathbf{r} : \mathtt{m}\langle\sigma\rangle \restriction \mathbf{p} = \begin{cases} \mathbf{r}!_{\mathsf{t}}\mathtt{m}\langle\sigma\rangle & \text{if } \mathbf{p} = \mathbf{q} \\ \mathbf{q}?_{\mathsf{t}}\mathtt{m}\langle\sigma\rangle & \text{if } \mathbf{p} = \mathbf{r} \\ \mathbf{skip} & \text{otherwise} \end{cases}$$

$$\mathbf{q}\downarrow \mathsf{t}(\!|\mathsf{C}|\!)\langle\sigma\rangle \restriction \mathbf{p} = \begin{cases} \mathsf{Put}\ \mathsf{t}(\!|\mathsf{C}|\!)\langle\sigma\rangle & \text{if } \mathbf{p} = \mathbf{q} \wedge \sigma(\mathbf{p})(\mathsf{Active}(\mathsf{t}), \mathit{SState}) \\ \mathsf{React}\ \mathsf{t}'\langle\sigma\rangle & \text{if } \mathbf{p} \neq \mathbf{q} \wedge \sigma(\mathbf{p})(\mathsf{Susp}, \mathit{SState}) \wedge (\mathsf{t}, \mathsf{t}') \in \mathit{SState} \\ \mathbf{skip} & \text{if } \mathbf{p} \neq \mathbf{q} \wedge \sigma(\mathbf{p})(\mathsf{Susp}, \mathit{SState}) \wedge \nexists \mathsf{t}'.\ (\mathsf{t}, \mathsf{t}') \in \mathit{SState} \end{cases}$$

$$\mathbf{q}\uparrow \mathsf{t}(\!|\mathsf{C}|\!)\langle\sigma\rangle \restriction \mathbf{p} = \begin{cases} \mathsf{Get}\ \mathsf{t}(\!|\mathsf{C}|\!)\langle\sigma\rangle & \text{if } \mathbf{p} = \mathbf{q} \wedge \sigma(\mathbf{p}) = (\mathsf{Active}(\mathsf{t}'), \mathit{SState}) \\ \mathbf{skip} & \text{otherwise} \end{cases}$$

$$\mathsf{Rel}(\mathbf{q}, \mathsf{t}) \restriction \mathbf{p} = \begin{cases} \mathsf{Susp}(\mathsf{t}', \mathsf{t})\langle\sigma\rangle & \text{if } \mathbf{p} = \mathbf{q} \wedge \sigma(\mathbf{p}) = (\mathsf{Active}(\mathsf{t}'), \mathit{SState}) \\ \mathbf{skip} & \text{if } \mathbf{p} \neq \mathbf{q} \end{cases}$$

$$\mathbf{q}\{\mathbf{G}_i\}_{i\in I}\langle\sigma\rangle \restriction \mathbf{p} = \begin{cases} \oplus\{\mathbf{G}_i \restriction \mathbf{p}\langle\sigma\rangle\}_{i\in I} & \text{if } \mathbf{p} = \mathbf{q} \wedge \sigma(\mathbf{p}) = (\mathsf{Active}(\mathsf{t}), \mathit{SState}) \\ \&_{\mathsf{t}}\{\mathbf{G}_i \restriction \mathbf{p}\langle\sigma\rangle\}_{i\in I} & \text{if } \mathbf{p} \neq \mathbf{q} \wedge \sigma(\mathbf{q}) = (\mathsf{Active}(\mathsf{t}), \mathit{SState}) \end{cases}$$

$$(\mathbf{G})^*\langle\sigma\rangle \restriction \mathbf{p} = \begin{cases} (\mathbf{L})^*\langle\sigma\rangle & \text{if } \mathbf{G} \restriction \mathbf{p} = \mathbf{L} \neq \mathbf{skip} \\ \mathbf{skip} & \text{otherwise} \end{cases}$$

$$(\mathbf{G}_1\ .\ \mathbf{G}_2) \restriction \mathbf{p} = (\mathbf{G}_1 \restriction \mathbf{p})\ .\ (\mathbf{G}_2 \restriction \mathbf{p}) \qquad \mathbf{skip} \restriction \mathbf{p} = \mathbf{skip}$$

Figure 4 Projection of global type on roles.

repeats the projection of the inner part if it performs some action. Otherwise, the repetition is replaced with an empty action. Finally, projection of sequential composition is sequential composition of the projected types. We assume that structural congruence is used to remove superfluous empty actions and branching.

3.2.3 Local Projection

Local projection generates a method-local type from an object-local type for each tracked future. Each tracked future is introduced by a call action, so we can easily connect method-local types to methods. For simplicity, we demand that each method has only one type.

Local projection must invert the relation between passive choice and synchronization. A global type specifies first the choice and marks the future of the choosing role during global projection. Afterwards, the future is resolved and may be synchronized upon. Locally, however, the method synchronizes *first* and *then* branches depending on the read value. Local projection handles this by pulling out the prefix of all branches from a passive choice up to the synchronization action over the choosing future.

▶ **Definition 12.** *Let* t *be a tracked future and* $\mathbf{L}_i$ *a set of object-local types. The prefix for* t *of some object-local* $\mathbf{L}$ *is defined as the shortest type* $\mathbf{L}_{\mathsf{t}}$ *that ends in* Get t(|C|)*: The function* $\mathsf{split}_{\mathsf{t}}(\mathbf{L})$ *returns the prefix and the remaining postfix of a type.*

$$\mathsf{split}_{\mathsf{t}}(\mathbf{L}) = (\mathbf{L}_{\mathsf{head}}, \mathsf{C}, \mathbf{L}_{\mathsf{tail}}) \textit{ such that}$$
$$\mathbf{L}_{\mathsf{head}} = \widehat{\mathbf{L}}\ .\ \mathsf{Get}\ \mathsf{t},\ \widehat{\mathbf{L}} \textit{ contains no } \mathsf{Get}\ \mathsf{t}, \textit{ and } \mathbf{L} \equiv \widehat{\mathbf{L}}\ .\ \mathsf{Get}\ \mathsf{t}(\!|\mathsf{C}|\!)\ .\ \mathbf{L}_{\mathsf{tail}}$$

The function $\mathsf{split}_{\mathsf{t}}(\{\mathbf{L}_i\}_{i\in I})$ *returns the common prefix and the remaining postfixes of all input types. Note that the function may be undefined.*

$$\mathsf{split}_{\mathsf{t}}(\{\mathbf{L}_i\}_{i\in I}) = (\mathbf{L}_{\mathsf{head}}, \{(\mathsf{C}_i, \mathbf{L}^i_{\mathsf{tail}}\})) \textit{ such that } \mathsf{split}_{\mathsf{t}}(\mathbf{L}_i) = (\mathbf{L}_{\mathsf{head}}, \mathsf{C}_i, \mathbf{L}^i_{\mathsf{tail}})$$

$$
\begin{aligned}
\mathbf{L}\langle\sigma\rangle \upharpoonright^{\mathbf{p}} \mathsf{t} &= \mathbf{L} \text{ if } \sigma(\mathbf{p}) = ((\mathsf{Active}(\mathsf{t}), \mathit{SState}) \\
&\qquad\qquad \text{and } \mathbf{L} \in \{\mathbf{p}!_{\mathsf{t}'}\mathtt{m},\ \mathsf{Put}\ \mathsf{t}'(\!|\mathsf{C}|\!),\ \mathsf{Get}\ \mathsf{t}'(\!|\mathsf{C}|\!),\ \mathsf{Susp}(\mathsf{t}',\mathsf{t}''),\ \mathbf{skip}\} \\
\mathbf{p}?_{\mathsf{t}'}\mathtt{m} \upharpoonright^{\mathbf{p}} \mathsf{t} &= \mathsf{React}\ \mathsf{t}' \upharpoonright^{\mathbf{p}} \mathsf{t} = \mathbf{skip} \\
(\mathbf{L}_1\ .\ \mathbf{L}_2) \upharpoonright^{\mathbf{p}} \mathsf{t} &= (\mathbf{L}_1) \upharpoonright^{\mathbf{p}} \mathsf{t}\ .\ (\mathbf{L}_2) \upharpoonright^{\mathbf{p}} \mathsf{t} \\
(\mathbf{L})^* \upharpoonright^{\mathbf{p}} \mathsf{t} &= \begin{cases} \mathbf{L} \upharpoonright^{\mathbf{p}} \mathsf{t} & \text{if } \mathsf{t} \text{ is introduced within } \mathbf{L} \\ (\mathbf{L}')^* & \text{if } \mathbf{L} \upharpoonright^{\mathbf{p}} \mathsf{t} = \mathbf{L}' \neq \mathbf{skip} \text{ and } \mathsf{t} \text{ is not introduced within } \mathbf{L} \\ \mathbf{skip} & \text{otherwise} \end{cases} \\
\oplus\{\mathbf{L}_i\}_{i\in I} \upharpoonright^{\mathbf{p}} \mathsf{t} &= \oplus\{\mathbf{L}_i \upharpoonright^{\mathbf{p}} \mathsf{t}\}_{i\in I} \\
\&_{\mathsf{t}'}\{\mathbf{L}_i\}_{i\in I} \upharpoonright^{\mathbf{p}} \mathsf{t} &= \begin{cases} \mathbf{L} \upharpoonright^{\mathbf{p}} \mathsf{t}\ .\ \&_{\mathsf{t}'}\{\mathsf{C}_i : \widehat{\mathbf{L}_i} \upharpoonright^{\mathbf{p}} \mathsf{t}\}_{i\in I} & \text{if } \mathsf{split}_{\mathsf{t}'}(\{\mathbf{L}_i\}_{i\in I}) = (\mathbf{L}, \{(\mathsf{C}_i, \widehat{\mathbf{L}_i})\}) \\ \&_{\mathsf{t}'}\{\mathbf{L}_i\}_{i\in I} \upharpoonright^{\mathbf{p}} \mathsf{t} = \mathbf{L}_j \upharpoonright^{\mathbf{p}} \mathsf{t} & \text{if } j \in J \text{ and } \mathsf{t} \text{ is introduced in } \mathbf{L}_j \end{cases}
\end{aligned}
$$

Figure 5 Projection of local types on tracked futures.

Projection $\mathbf{L}\langle\sigma\rangle \upharpoonright^{\mathbf{p}} \mathsf{t}$ of an annotated local type $\mathbf{L}\langle\sigma\rangle$ on t for role $\mathbf{p}$ is given in Fig. 5. It removes receiving and reactivation actions, is the identity on any other non-composed action and propagates on sequential composition and active choice. For passive choice, the above split is applied, unless the projection future is introduced in only one branch. Repetitions outside a single method run are removed.

3.2.4 Scheduling Type

Given a projected object-local type $\mathbf{L}$, the scheduling type $\mathcal{S}(\mathbf{L})$ is generated by replacing all termination, synchronization, suspension and sending actions with **skip** and using structural congruence (see Fig. 6) to simplify the result.

3.3 Locally Static

Method-local types are checked statically. This ensures that *if* every process is scheduled correctly, then the process will perform its local view on the protocol correctly. Before we present the type system itself, we define typing contexts and auxiliary functions.

The subtype relations $<$, $\leq$ and structural congruence are standard, see Fig. 6. Structural congruence allows to add and remove **skip** actions. An active choice with a single branch can be simplified to the content of the branch. The interesting rules for subtyping are the ones for branching: Active branching may drop branches, as the implementing role may never take a subset of its possible choices. Its dual, passive branching, may add branches instead.

We use two typing contexts: Δ maps locations (fields and variables) to roles, Γ maps tracked futures to pairs of locations or the symbol $\perp$. the Δ context ensures that a method interacts with the correct endpoints, while Γ keeps track of futures and their read values. We use some auxiliary functions and predicates:

- The function $\Gamma^{\mathfrak{A}}$ removes all fields from the pairs in the image of Γ.
- The function $\mathsf{constr}(\mathtt{e})$ returns the outermost constructor of expression $\mathtt{e}$.
- The function $\mathsf{def}(\mathtt{C})$ returns the declaration of class $\mathtt{C}$.
- The predicate $\mathsf{inter}(\mathtt{s}, \Gamma)$ holds if the statement $\mathtt{s}$ contains no **get**, no **return**, no **await**, and writes into no location that is in a pair in the image of Γ.

$$
\begin{array}{rl}
\oplus\{\mathbf{L}_i\}_{i\in I} < \oplus\{\mathbf{L}_i\}_{i\in I\cup J} \qquad \&_{\mathsf{t}}\{\mathsf{C}_i : \mathbf{L}_i\}_{i\in I} > \&_{\mathsf{t}}\{\mathsf{C}_i : \mathbf{L}_i\}_{i\in I\cup J} & \\
(\mathbf{L})^* < (\widehat{\mathbf{L}})^* & \text{if } \mathbf{L} < \widehat{\mathbf{L}} \\
\mathbf{L}_1.\mathbf{L}_2 < \widehat{\mathbf{L}_1}.\widehat{\mathbf{L}_2} & \text{if } \mathbf{L}_i < \widehat{\mathbf{L}_i} \\
\oplus\{\mathbf{L}_i\}_{i\in I} < \oplus\{\widehat{\mathbf{L}_i}\}_{i\in I} & \text{if } \mathbf{L}_i < \widehat{\mathbf{L}_i} \\
\&_{\mathsf{t}}\{\mathsf{C}_i : \mathbf{L}_i\}_{i\in I} < \&_{\mathsf{t}}\{\mathsf{C}_i : \widehat{\mathbf{L}_i}\}_{i\in I} & \text{if } \mathbf{L}_i < \widehat{\mathbf{L}_i} \\
\mathbf{L} \equiv \oplus\{\mathbf{L}\} \quad \mathbf{L} \equiv \mathbf{skip}\ .\ \mathbf{L} \quad \mathbf{L} \equiv \mathbf{L}\ .\ \mathbf{skip} & \\
\mathbf{L}\ .\ \oplus\{\mathbf{L}_i\}_{i\in I} \equiv \oplus\{\mathbf{L}\ .\ \mathbf{L}_i\}_{i\in I} \qquad \oplus\{\mathbf{L}, \mathbf{skip}_i\}_{i\in I} \equiv \mathbf{L} &
\end{array}
$$

Figure 6 Subtype relation and structural congruence of method-local types.

- The predicate $\mathbf{p} \in \mathbf{G}$ or $\mathbf{p} \in \mathbf{L}$ holds if the role $\mathbf{p}$ occurs in the type.
- The predicate $\mathtt{e} \in \mathbf{im}\Gamma$ holds if the location $\mathtt{e}$ is in any pair in the image of Γ.

The type system is shown in Fig. 7. Rule T-**main** checks that the main block sets up the session correctly: Each role is assigned to exactly one object and the corresponding class is checked against the projected type on this role. Also, each parameter of a class is assigned such that the passed variable has the correct role (the f_{ij} are the fields declared in $\mathsf{def}(\mathtt{C}_i)$). Lastly, the sole called method is correctly specified and called on the correct object. The rule T-**class** checks that each role needed for the object-local type is available in some field and checks each method against its method-local type.

$$
\text{T-}\textbf{main}\ \frac{\begin{array}{c}\forall i.\ \exists \mathbf{p}\in\mathbf{G}.\ \Delta(\mathtt{v}_i) = \mathbf{p} \qquad \forall \mathbf{p}\in\mathbf{G}.\ \exists i.\ \Delta(\mathtt{v}_i) = \mathbf{p} \\ \Delta_i(f_{ij}) = \Delta(\mathtt{v}_{ij}) \qquad \Delta_i \vdash \mathsf{def}(\mathtt{C}_i)\ :\ \mathbf{G} \upharpoonright \Delta(\mathtt{v}_i) \qquad \Delta(\mathtt{v}_k) = \mathbf{p}\end{array}}{\vdash \{\mathtt{I}_i\ \mathtt{v}_i\ \mathtt{=}\ \textbf{new}\ \mathtt{C}_i(\mathtt{v}_{ij})\mathtt{;v}_k\mathtt{!m();}\}\ :\ \mathbf{0} \xrightarrow{\mathsf{t}} \mathbf{p}\mathtt{:m}\ .\ \mathbf{G}}
$$

$$
\text{T-}\textbf{class}\ \frac{\forall \mathbf{p}\in\mathbf{L}.\ \exists i.\ \Delta(\mathtt{f}_i) = \mathbf{p} \qquad \Delta,\emptyset \vdash \mathtt{s}_k\ :\ \mathbf{L} \upharpoonright^{\mathbf{p}} \mathsf{t} \qquad \mathtt{m}_k \text{ is the method of } \mathsf{t} \text{ in } \mathbf{L}}{\Delta \vdash \textbf{class}\ \mathtt{C(I}_i\ \mathtt{f}_i\mathtt{)\{T}_j\ \mathtt{f}_j\ \mathtt{=}\ \mathtt{e}_j\mathtt{;}\ \mathtt{T}_k\ \mathtt{m}_k\mathtt{(T}_{kl}\ \mathtt{v}_{kl}\mathtt{)\{s}_k\mathtt{\}\}}\ :\ \mathbf{L}}
$$

$$
\text{T-}\leq\ \frac{\Delta,\Gamma\vdash \mathtt{s}\ :\ \mathbf{L} \qquad \mathbf{L}\equiv\widehat{\mathbf{L}}'\leq\widehat{\mathbf{L}}}{\Delta,\Gamma\vdash\mathtt{s}\ :\ \widehat{\mathbf{L}}}
\qquad
\text{T-;}\ \frac{\Delta,\Gamma\vdash\mathtt{s}_2\ :\ \mathbf{L} \qquad \mathsf{inter}(\mathtt{s}_1,\Gamma)}{\Delta,\Gamma\vdash\mathtt{s}_1\mathtt{;s}_2\ :\ \mathbf{L}}
$$

$$
\text{T-}\textbf{return}\ \frac{\mathsf{constr}(\mathsf{e}) = \mathsf{C}}{\Delta,\Gamma\vdash\textbf{return}\ \mathtt{e;}\ :\ \mathsf{Put}\ \mathsf{t}(\!|\mathsf{C}|\!)}
\qquad
\text{T-}\textbf{skip}\ \frac{}{\Delta,\Gamma\vdash\textbf{skip}\ :\ \mathbf{skip}}
$$

$$
\text{T-}\textbf{if}\ \frac{\begin{array}{c}\Delta,\Gamma\vdash\mathtt{s}_1\mathtt{;s}_3\ :\ \mathbf{L}\\ \Delta,\Gamma\vdash\mathtt{s}_2\mathtt{;s}_3\ :\ \mathbf{L}\end{array}}{\Delta,\Gamma\vdash\textbf{if}\mathtt{(e)\{s}_1\mathtt{\}}\textbf{else}\mathtt{\{s}_2\mathtt{\}s}_3\ :\ \mathbf{L}}
\qquad
\text{T-}\textbf{while}\ \frac{\begin{array}{c}\Delta,\widetilde{\Gamma}\vdash\mathtt{s}_1\ :\ \mathbf{L}_1\\ \Delta,\widetilde{\Gamma}\vdash\mathtt{s}_2\ :\ \mathbf{L}_2\end{array}}{\Delta,\Gamma\vdash\textbf{while}\mathtt{(e)\{s}_1\mathtt{\}s}_2\ :\ (\mathbf{L}_1)^*.\mathbf{L}_2}
$$

$$
\text{T-}\textbf{await}\ \frac{\Delta,\Gamma^{\mathfrak{A}}\vdash\mathtt{s}\ :\ \mathbf{L} \qquad \Gamma(\mathsf{t}') = (\mathsf{e},\_)}{\Delta,\Gamma\vdash\textbf{await}\ \mathtt{e;}\ \mathtt{s}\ :\ \mathsf{Susp}(\mathsf{t},\mathsf{t}').\mathbf{L}}
$$

$$
\text{T-}\textbf{get}\ \frac{\Gamma(\mathsf{t}) = (\mathsf{e}_2,\_) \qquad \Delta,\Gamma[\mathsf{t}\mapsto(\mathsf{e}_2,\mathsf{e}_1)]\vdash\mathtt{s}\ :\ \mathbf{L} \qquad \mathsf{e}_1\not\in\mathbf{im}\Gamma}{\Delta,\Gamma\vdash\mathtt{e}_1\ \mathtt{=}\ \mathtt{e}_2\mathtt{.}\textbf{get}\mathtt{;}\ \mathtt{s}\ :\ \mathsf{Get}\ \mathsf{t}.\mathbf{L}}
$$

$$
\text{T-!}\ \frac{\Delta,\Gamma[\mathsf{t}\mapsto(\mathsf{e}_1,\bot)]\vdash\mathtt{s}\ :\ \mathbf{L} \qquad \Delta(\mathsf{e}_2) = \mathbf{p} \qquad \mathsf{e}_1\not\in\mathbf{im}\Gamma}{\Delta,\Gamma\vdash\mathtt{e}_1\ \mathtt{=}\ \mathtt{e}_2\mathtt{!m(e);}\ \mathtt{s}\ :\ \mathbf{p}!_{\mathsf{t}}\mathtt{m}.\mathbf{L}}
$$

$$
\text{T-}\textbf{case}\ \frac{\mathsf{C}_i = \mathsf{C}_j \rightarrow \Delta,\Gamma\vdash\mathtt{s}_i\ :\ \mathbf{L}_j.\mathbf{L} \qquad \forall j.\ \exists i.\ \mathsf{C}_i = \mathsf{C}_j \qquad \Gamma(\mathsf{t}) = (\_,\mathsf{e})}{\Delta,\Gamma\vdash\textbf{case}\mathtt{(e)\{}\mathsf{C}_i\mathtt{=>s}_i\mathtt{\}}_{i\in I}\mathtt{s}\ :\ \&_{\mathsf{t}}\{\mathsf{C}_j : \mathbf{L}_j\}.\mathbf{L}}
$$

Figure 7 Static Type System.

Rule T-$\leq$ is used for structural congruence and sub-typing. The construction of a syntax-directed variant of the type system without a special rule for subtying is standard. Rule T-`;` drops a prefix that performs no communication and modifies no location stored in Γ. Rule T-**return** checks that the sole remaining action is a Put action and that the correct constructor is returned. Rule T-**skip** closes the proof if the empty program **skip** is left and no further action is required. This is needed to typecheck loop bodies, where we can always add **skip** at the end. Rule T-**if** splits the derivation into two branches. The type is not changed. Rule T-**while** checks a loop against the Kleene star. The context $\widetilde{\Gamma}$ removes all fields and variables modified in the loop body. Rule T-**await** checks that the correct future is synchronized on and removes all fields from the context. Rule T-**get** checks that the correct future is read and stores the information where the read value is available in the context. It ensures that no relevant read value or future is overwritten. Rule T-`!` checks that the correct method on the correct role is called and stores the information where the future is available in the context. It ensures that no other relevant read value or future is overwritten. Rule T-**case** checks a **case** statement against a passive choice by mapping each branch of the statement against some branch of the type. It is ensured that for every specified choice an implemented branch exists and that the read value is indeed stemming from the future containing the choice.

A rule for assignments to copy futures or their read values is easily added, but requires to keep track of a pair of sets of locations and, for simplicity, we refrain from introducing this.

3.4 Globally Dynamic

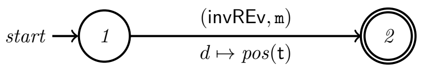
The globally dynamic part consists of two steps: first, we translate an object-local type into a session automaton, then we translate the session automaton into a user-defined scheduler.

3.4.1 Automaton Extraction

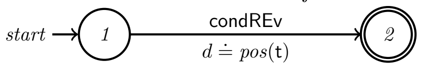
The structure of the translation follows the standard translation of regular expressions into finite automata.

▶ **Definition 13.** *Let* $\mathbf{L}$ *be a projected object-local type with* k *tracked futures. Let* $pos(\mathtt{t})$ *be the register assigned to* $\mathtt{t}$*. The translation of* $\mathbf{L}$ *into a* k*-register session automaton is denoted* $\mathcal{A}(\mathbf{L})$ *and defined as follows:*

- *A receiving type* $\mathbf{p}?_{\mathtt{t}}\mathtt{m}$ *is translated into an automaton with two states and a single transition that reads* $\mathsf{invREv}, \mathtt{m}$ *and stores the read future in* $pos(\mathtt{t})$*:*



- *A reactivation type* $\mathsf{React}\ (\mathtt{t})$ *is translated into an automaton with two states and a single transition that reads* $\mathsf{condREv}$ *and matches the read future with the one stored in* $pos(\mathtt{t})$*.*



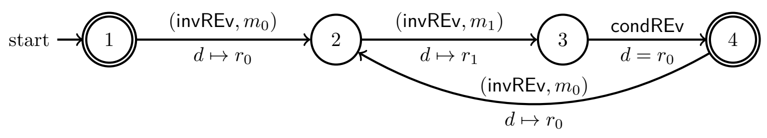
- *Branching, sequence and repetition are the standard translations of alternative, concatenation and Kleene star into finite automata.*

After this construction, standard ϵ*-transition elimination is performed.*

▶ **Example 14.** Consider the following scheduling type [29]:

$$\mathbf{L} = \left(\mathbf{p}?_{\mathtt{t}_0}\mathtt{m}_0\ .\ \mathbf{p}?_{\mathtt{t}_1}\mathtt{m}_1\ .\ \mathsf{React}\ (\mathtt{t}_0)\right)^*$$

Its translation $\mathcal{A}(\mathbf{L})$ is as follows (the translation yields an ϵ-transition from state 4 to state 1, which is eliminated to give the depicted automaton):



For formal soundness arguments, we again refer to [20]. Intuitively, the extraction is sound because the language accepted by the automaton is the same language as the one generated by the object-local type. Not every extracted session automaton is deterministic, because the input object-local type may not be deterministic, for example:

$$\&_{\mathsf{t}} \left\{ \begin{array}{l} \mathbf{p}?\mathtt{m} \\ \mathbf{p}?\mathtt{m}\ .\ \mathbf{q}?\mathtt{n} \end{array} \right\}$$

After receiving a call on `m`, this type cannot predict which branch to take. We do allow non-deterministic schedulers, but the implementation issues a warning. A simple syntax check on the automaton can exclude them.

3.4.2 Translation and Integration

Given a session automaton, we can finally extract a user-defined scheduler and add instrumentation code to ensure correctness.

▶ **Definition 15.** *Let* `C` *be a class that is checked against an object-local type that is transformed to a scheduling type* $\mathbf{L}$*. The instrumented class* $\mathtt{C}^{\mathcal{I}}$ *is constructed as follows:*

- *We add a field* `Int q = 0;` *that models the current state of the scheduling automaton.*
- *For each register* r_i *we add a field "*`Maybe<Fut<Any>> ri = Nothing;`*".*
- *The scheduler is as in Def. 16.*
- *For each method* `m` *we collect all transitions* $(q_i, (\mathsf{invREv}, \mathtt{m}), q_{i'})_{i \in I}$ *with written register reg*(i) *and add the following as the first statement of* `m`*:*

```
case this.q {
  q_{i_1} => this.r reg(i_1) = Just(destiny); this.q = q_{i'_1};
  ⋮
  q_{i_m} => this.r reg(i_m) = Just(destiny); this.q = q_{i'_m};
}
```

  This statement saves its future in the given register and updates the automaton state. The generated scheduler ensures that no default branch is needed.
- *For each class* `C` *we collect all transitions* $(q_i, (\mathsf{condREv}), q_{i'})_{i \in I}$ *with read register reg*(i) *and add the following as the first statement after each* **await** *statement in any method:*

```
case this.q { q_{i_1} => this.q = q_{i'_1}; ... q_{i_m} => this.q = q_{i'_m}; }
```

  Again, the generated scheduler ensures that no default branch is needed and the registers do not need to be checked against `destiny`*.*

▶ **Definition 16.** *The generated scheduler ensures that the initializing method with the hidden name* `.init()` *is always executed first. The function* `filter` *is one of the higher-order functions in ABS and takes a function of the form* `(params) => code` *as its first parameter and a list as its second.*

```
def Maybe<Process> scheduler(List<Process> list,
                             Int q,
                             Maybe<Fut<Any>> r1,
                             Maybe<Fut<Any>> r2) =
   if ( filter((Process p) => method(p) == ".init")(queue) != Nil )
       headOrNothing(filter((Process p) => method(p) == ".init")(queue))
   else case q {
     1 => headOrNothing(
        filter((Process p) => contains(set["publish"],method(p)))(list));
     2 => headOrNothing(
        filter((Process p) => contains(set["request"],method(p)))(list));
   }
```

Figure 8 Scheduler generated from Ex. 11.

```
def Maybe<Process> scheduler(List<Process> list, Int q,
                             Maybe<Fut<Any>> r_1, ..., Maybe<Fut<Any>> r_n) =
   if( filter((Process p) => method(p) == ".init")(queue) != Nil )
        headOrNothing(filter((Process p) => method(p) == ".init")(queue))
   else scheduler_body(list, q, r_1, ..., r_n);
```

After executing the initializer, the scheduler makes a case distinction over the states $1, \ldots, m$ of the scheduling automaton:

```
def Maybe<Process> scheduler_body(List<Process> list, Int q,
                                  Maybe<Fut<Any>> r_1, ..., Maybe<Fut<Any>> r_n) =
  case q { 1 => transition_1; ... m => transition_m; }
```

The transition transition$_i$ *from a state i is modeled as follows: Let* m$_1, \ldots,$ m$_{n_1}$ *be the method names that have outgoing transitions from i labeled with* invREv*. Let* r'$_1, \ldots,$ r'$_{n_2}$ *be the registers that the read future is compared with in outgoing transitions from i labeled with* condREv*. The first case checks that the future is allowed and not yet stored, the second case checks that the future is in one of the registers.*

```
headOrNothing(filter((Process p) =>
(contains(set[m_1,...,m_{n_1}],method(p)) && !contains(set[r_1,...,r_n],destinyOf(p)))
|| contains(set[r'_1,...,r'_{n_2}],destinyOf(p))
)(list))
```

We return the first process that is in the list and matches, a random scheduler is a straightforward modification.

▶ **Example 17.** The (beautified) scheduler generated from Ex. 11 is shown in Fig. 8:

3.5 Soundness and Stateful Session Types

Soundness. Soundness of the type system follows directly from the soundness theorem given for the original, purely static systems [27, 28]:

▶ **Theorem 18.** *Let* Prgm *be a well-typed ABS program and* $\mathbf{GP}$ *a global protocol. If* $\vdash$ Prgm : $\mathbf{GP}$ *and every object is instrumented with the scheduler type derived by the* $\vdash$ *relation, then every terminating and non-deadlocking run of* Prgm *has a trace where the communication events for each object are in the same order as specified in* $\mathbf{GP}$*.*

Our notion of soundness is not based on subject reduction and progress. Soundness of our system is *only* concerned with protocol adherence, not with deadlock freedom, as discussed above. Adding deadlock checks in session types complicates the system further [28] for little gain, as external tools can be used. We assume that the data types have been checked, so there is no need for a progress theorem. It is similar to *session fidelity* [22], which expresses the same intuition in terms of operational semantics.

Neither do we use a subject reduction theorem. Instead, we give a denotational semantics to session types and regard them as specifications of traces in monadic second-order logic: Any type $\mathbf{GP}$ can be translated into a formula $\mathbb{C}(\mathbf{GP})$ expressing that the communication events for each object are in the same order as specified in $\mathbf{GP}$. Soundness is then a *model-theoretic* notion that every trace $\mathbf{tr}$ generated by $\mathbf{GP}$ is a model of $\mathbb{C}(\mathbf{GP})$:

$$\vdash \mathsf{Prgm} : \mathbf{GP} \rightarrow \forall \mathbf{tr}.\ \mathsf{Prgm} \Downarrow \mathbf{tr} \rightarrow \mathbf{tr} \models \mathbb{C}(\mathbf{GP})$$

This model-theoretic treatment of session types allows an elegant connection to symbolic execution and dynamic logic [27] at the cost of an elaborate semantics [14] which we refrain to introduce for space reasons. This semantics is based on merging of local traces, which inhibits us from giving a straightfoward subject reduction theorem.

Stateful Session Types. So far, our session types do not constrain the execution state or passed data, except the outermost constructor of return values. We implemented an extension of the presented system, where each global call action is annotated with a property. This annotation is preserved during object-local projection and moved to the termination action during projection on a tracked future. Regarding instrumentiation, it results in a simple **assert** statement for the dynamic check.

▶ **Example 19.** We specify that a call of role $\mathbf{p}$ to a method `m` results in a postcondition that ensures the return value being larger than field `f`:

$$\ldots\ .\ \mathbf{p} \xrightarrow{\mathsf{t}} \mathbf{q}{:}\mathtt{m}(\!|\mathtt{this.f\ <\ result}|\!)\ .\ \ldots$$

The return value is saved in a dedicated variable `result` and an assert is added afterwards. If the final statement was "`return e;`" before, it now is

```
Int result = e; assert(this.f < result); return result;
```

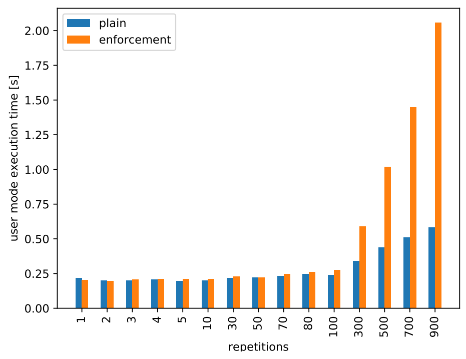
If it depends on the state of the scheduler which postcondition has to be checked, a **case** statement over the possible values of `q` is added. This approach is slightly less expressive than other stateful session types for AO [26, 28], but has the benefit that there is no need to translate first-order logic formulas into expressions.

4 Implementation and Evaluation

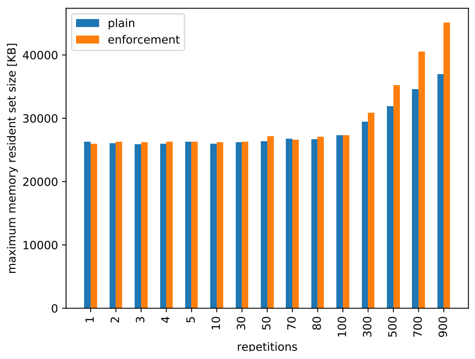
Our system is implemented on top of a slightly modified[5] version of the `ABS` compiler [43, 39]. Source code and all examples are accessible at `https://github.com/ahbnr/SessionTypeABS`.

As discussed, we do not handle full `ABS` and demand that the main block initializes a whole session, each interface plays exactly one role and no objects are created after initialization. The session type is specified in an ASCII variant of Def. 4 in a separate file alongside the other

[5] Blocking schedulers and access to the future of a process are not yet part of the master branch of `ABS`.



■ **Figure 9** Execution times of the unmodified (blue) and modified (orange) model for different amounts of repetitions.



■ **Figure 10** Maximum memory resident set size of the unmodified (blue) and modified (orange) model for different ammounts of repetitions.

model source files. The `ABS` compiler is used for parsing and (data-)typechecking the input model. The AST is then used for the static check and enriched with the instrumentation from the scheduling type. The resulting AST is passed back to the `ABS` compiler, which parses and typechecks it again.

We evaluate the impact of our modifications on the performance of Erlang-based simulations (the standard backend) of `ABS` models. The experiments are performed on a synthetic benchmark, where one object implementing the role **p** repeatedly calls two methods on another object of role **q** in a fixed order. The session is specified by the this global type:

$$\mathbf{0} \xrightarrow{\mathtt{t}} \mathbf{p}:\mathtt{init}\ .\ \left(\mathbf{p} \xrightarrow{\mathtt{t_{m1}}} \mathbf{q}:\mathtt{m1}\ .\ \mathbf{q}\downarrow \mathtt{t_{m1}}\ .\ \mathbf{p}\xrightarrow{\mathtt{t_{m2}}}\mathbf{q}:\mathtt{m2}\ .\ \mathbf{q}\downarrow\mathtt{t_{m2}}\right)^{*}\ .\ \mathbf{p}\downarrow\mathtt{t}$$

All reported data resulted from executing the model multiple times and averaging the measurements. Reported *execution times* designate the required run time in user-mode of the Erlang simulation of a model until termination on a Arch Linux system running Kernel 5.3.7 with a i5-4300U@2.9GHz CPU and 4GiB RAM.

Effect of Increased Object Communication. By changing the number of times the repeatable section of the session type is executed, we observe the behavior of the model simulation when the number of calls from **p** to **q** increases. We observe that the user-mode execution time of the simulations is nearly constant and mostly equivalent for the modified and unmodified version of the model for up to 100 repetitions, see Fig. 9. For higher numbers of repetitions, execution time increases for both version, but execution time of the modified model grows to increasing multiples of the execution time of the unmodified one. The maximum memory resident set size of the Erlang processes develops similarly, see Fig. 10, although the memory size of the modified model does not grow as rapidly as the execution time.

Comparison to Manual Synchronization. Instead of letting the generated schedulers enforce the execution order of methods, we now require **p** and **q** to synchronize every call by inserting an **await**-statement after each interaction. Even though execution time of the unmodified and modified model still increases for a high number of repetitions, there is now little difference between them, see Fig. 11. The overhead of synchronization is roughly equivalent to or lower than the version relying on the generated schedulers.

Testing the Reordering Capabilities of the Scheduler: In the previous experiments the scheduler never delayed activating a process, because there was always one in the queue which could immediately be scheduled. We now disable static verification, deliberately reverse the

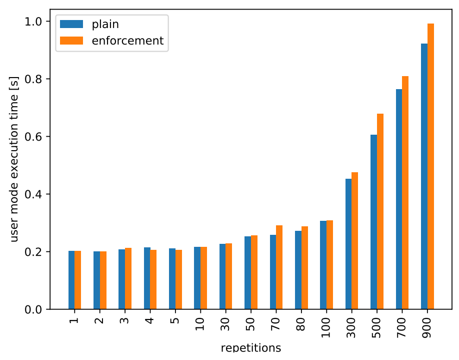


Figure 11 User-mode execution times when using **await** statements. Unmodified model in blue, modified model in orange.

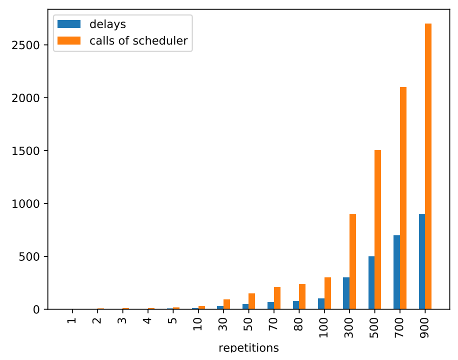


Figure 12 The number of times a scheduler has been invoked (orange) in contrast to the number of times it could not activate any waiting process (blue).

calls in the model source and put `duration` statements after each call, causing a delay in the execution.[6] We do not use synchronization and calls always arrive out of order at **q** and with enough inactivity in between them so that the scheduler of **q** frequently has to delay activation of a process until an acceptable one is available. Here, the modified and unmodified model always complete in almost the same execution time, presumably since the `duration` statements induce enough idle time to contain the overhead of the schedulers. However, we now observe that the scheduler successfully delays and reorders calls, see Fig. 12.

Discussion. A certain overhead must always be expected from instrumentation, but we deem the observed overhead acceptable. The generated schedulers only result in noteworthy overhead when a large number of processes is in the object queue. We conjecture that this effect is mostly an artifact of how the queue is represented for the user-defined scheduler.

5 Related Work

There is a considerable number of papers combining static and dynamic verification, a complete overview is out of scope for this work. We refer to, for example, the introduction of Ahrendt et al. [2] and only review directly related approaches here.

The `StaRVOOrS` [1, 9] tool combines static and dynamic verification of Java programs as follows: First, it attempts to prove certain properties statically using deductive verification and then it transforms failed proofs into runtime monitors. The static analysis is used to ensure that as little as possible is checked dynamically. `StarVOOrS` distinguishes between data and control-flow properties. The static analysis is mainly reducing the need for the computationally heavy data properties (e.g., all values of an array are non-zero) as far as possible, while monitoring control-flow properties can be done statically.

Our approach can be seen from a similar perspective: the object scheduler is handling the control flow inside an object, while the added **assert** statements are handling data properties. The type checker ensures that inside a method, only data properties need to be checked at

[6] Explicit time behavior is realized in Timed `ABS` [5] and here only used for evaluation.

runtime. It is straightforward to see how the ongoing integration of Session Types into the `Crowbar` prover using Behavioral Program Logic [26] can be used to discard as superflous **assert** statements statically.

The literature on session types includes approaches that handle protocols as (partially) dynamic types or mix static and dynamic checks otherwise. The conceptually closest to our approach is by Bocchi et at. [6], who also use distributed runtime enforcement, but introduce new components (for example, a queue) to do so. Completely dynamic approaches to session types are available for the Python language [12] and an actor model [35]. Other, less related, approaches are:

- Gradual session types [23] transform a dynamically checked *dyadic* session type for *channels* gradually to a statically checked one during development. The dynamic check for linearity that is central to gradual session types for channels has no direct counterpart in our system for AO, because the projection mechanisms differ on a technical level.
- Certain combined approaches, e.g., for Scala [38], draw the line between static and dynamic by performing only the linearity check at runtime and any other check statically.

A further type-based approach is *typestate* [41]. In contrast to session types, it was developed mainly for OO imperative programs. Typestate models that an object can change its interface, i.e., the set of exposed methods, over time. This was done statically in the original work and was subsequently gradualized [42] to combine static and dynamic type checking. A variant of typestate for concurrent Java, developed by Gerbo & Padovani [17], dynamically reports violations after injecting monitoring code. The object scheduler in our approach can be seen as a variant of typestate, but it is *generated*, not specified.

Choreographies [8] bear similarity to session types, being global specifications with a projection mechanism. However, they are mainly used to *generate* code via a correctness-by-construction approach. This also combines static and dynamic aspects, but reverses the direction: instead of dynamically ensuring that the static checks are sound, it is statically ensured (by code generation) that the dynamic behavior is structured correctly. The distinction between static and dynamic parts becomes even more prominent in the work of **Gabbrielli** et al. [16, 36, 37], where *dynamic choreographies* are used to generate a dynamic structure to update the structure of the application or include of new participants.

6 Conclusion

What should be the takeaway message from this work? First, the formalism of session types, first developed in the context of the π-calculus, and so far mainly used in theoretical investigations, appears in our context as a rather versatile and surprisingly practical specification mechanism. It is easily conceivable to find a more user-friendly, less mathematical notation for the global types in Fig. 3 and add IDE support.

Second, with the runtime checking approach, session types for AO can form the theoretical basis for top-down development of *open* distributed systems (with cooperative concurrency).

Third, as shown here and in [27], session types integrate well with static checking of logical properties. The semantic link is a straightforward translation from session types into logic , while the type systems syntactically ensures to place assertions at suitable locations.

Future Work. We plan to adopt the `StaRVOOrS` approach to partially reduce the need for **assert** statements on method-local level. We are investigating the use of the product line mechanism of `ABS` [10] to add the monitors, instead of using manual code injection. Using product lines enables a uniform treatment of code injection in `ABS` and the injection and removal of runtime monitors at runtime [40]. Furthermore, we plan to investigate the use of Timed Session Types [34] for Timed `ABS` and Hybrid `ABS` [32].

References

1 Wolfgang Ahrendt, Jesús Mauricio Chimento, Gordon J. Pace, and Gerardo Schneider. Verifying data- and control-oriented properties combining static and runtime verification: theory and tools. *Formal Methods Syst. Des.*, 51(1):200–265, 2017.

2 Wolfgang Ahrendt, Marieke Huisman, Giles Reger, and Kristin Yvonne Rozier. A broader view on verification: From static to runtime and back (track summary). In *ISoLA (2)*, volume 11245 of *Lecture Notes in Computer Science*, pages 3–7. Springer, 2018.

3 Elvira Albert, Puri Arenas, Antonio Flores-Montoya, Samir Genaim, Miguel Gómez-Zamalloa, Enrique Martin-Martin, German Puebla, and Guillermo Román-Díez. SACO: static analyzer for concurrent objects. In Erika Ábrahám and Klaus Havelund, editors, *TACAS 2014*, volume 8413 of *Lecture Notes in Computer Science*, pages 562–567. Springer, 2014. doi:10.1007/978-3-642-54862-8_46.

4 Henry G. Baker and Carl E. Hewitt. The incremental garbage collection of processes. In *Proceeding of the Symposium on Artificial Intelligence Programming Languages*, number 12 in SIGPLAN Notices, page 11, August 1977.

5 Joakim Bjørk, Frank S. de Boer, Einar Broch Johnsen, Rudolf Schlatte, and Silvia Lizeth Tapia Tarifa. User-defined schedulers for real-time concurrent objects. *ISSE*, 9(1):29–43, 2013.

6 Laura Bocchi, Tzu-Chun Chen, Romain Demangeon, Kohei Honda, and Nobuko Yoshida. Monitoring networks through multiparty session types. *Theor. Comput. Sci.*, 669:33–58, 2017. doi:10.1016/j.tcs.2017.02.009.

7 Benedikt Bollig, Peter Habermehl, Martin Leucker, and Benjamin Monmege. A fresh approach to learning register automata. In Marie-Pierre Béal and Olivier Carton, editors, *Developments in Language Theory: 17th Intl. Conf. DLT, Marne-la-Vallée, France*, volume 7907 of *LNCS*, pages 118–130. Springer, 2013.

8 Marco Carbone, Kohei Honda, and Nobuko Yoshida. Structured communication-centered programming for web services. *ACM Trans. Program. Lang. Syst.*, 34(2):8:1–8:78, 2012. doi:10.1145/2220365.2220367.

9 Jesús Mauricio Chimento, Wolfgang Ahrendt, Gordon J. Pace, and Gerardo Schneider. Starvoors: A tool for combined static and runtime verification of java. In Ezio Bartocci and Rupak Majumdar, editors, *RV 2015*, volume 9333 of *Lecture Notes in Computer Science*, pages 297–305. Springer, 2015. doi:10.1007/978-3-319-23820-3_21.

10 Dave Clarke, Radu Muschevici, José Proença, Ina Schaefer, and Rudolf Schlatte. Variability modelling in the ABS language. In Bernhard K. Aichernig, Frank S. de Boer, and Marcello M. Bonsangue, editors, *FMCO 2010*, volume 6957 of *Lecture Notes in Computer Science*, pages 204–224. Springer, 2010. doi:10.1007/978-3-642-25271-6_11.

11 Frank S. de Boer, Vlad Serbanescu, Reiner Hähnle, Ludovic Henrio, Justine Rochas, Crystal Chang Din, Einar Broch Johnsen, Marjan Sirjani, Ehsan Khamespanah, Kiko Fernandez-Reyes, and Albert Mingkun Yang. A survey of active object languages. *ACM Comput. Surv.*, 50(5):76:1–76:39, 2017. doi:10.1145/3122848.

12 Romain Demangeon, Kohei Honda, Raymond Hu, Rumyana Neykova, and Nobuko Yoshida. Practical interruptible conversations: distributed dynamic verification with multiparty session types and python. *Formal Methods Syst. Des.*, 46(3):197–225, 2015. doi:10.1007/s10703-014-0218-8.

13 Crystal Chang Din, Richard Bubel, and Reiner Hähnle. Key-abs: A deductive verification tool for the concurrent modelling language ABS. In Amy P. Felty and Aart Middeldorp, editors, *Automated Deduction - CADE-25 - 25th International Conference on Automated Deduction, Berlin, Germany, August 1-7, 2015, Proceedings*, volume 9195 of *Lecture Notes in Computer Science*, pages 517–526. Springer, 2015. doi:10.1007/978-3-319-21401-6_35.

14 Crystal Chang Din, Reiner Hähnle, Einar Broch Johnsen, Ka I Pun, and Silvia Lizeth Tapia Tarifa. Locally abstract, globally concrete semantics of concurrent programming languages. In *TABLEAUX*, volume 10501 of *Lecture Notes in Computer Science*, pages 22–43. Springer, 2017.

15 Antonio Flores-Montoya, Elvira Albert, and Samir Genaim. May-happen-in-parallel based deadlock analysis for concurrent objects. In *FMOODS/FORTE*, volume 7892 of *LNCS*, pages 273–288. Springer, 2013.

16 Maurizio Gabbrielli, Saverio Giallorenzo, Ivan Lanese, and Jacopo Mauro. Guess who's coming: Runtime inclusion of participants in choreographies. In Mário S. Alvim, Kostas Chatzikokolakis, Carlos Olarte, and Frank Valencia, editors, *The Art of Modelling Computational Systems: A Journey from Logic and Concurrency to Security and Privacy - Essays Dedicated to Catuscia Palamidessi on the Occasion of Her 60th Birthday*, volume 11760 of *Lecture Notes in Computer Science*, pages 118–138. Springer, 2019. `doi:10.1007/978-3-030-31175-9_8`.

17 Rosita Gerbo and Luca Padovani. Concurrent typestate-oriented programming in java. In Francisco Martins and Dominic Orchard, editors, *Proceedings Programming Language Approaches to Concurrency- and Communication-cEntric Software, PLACES@ETAPS 2019, Prague, Czech Republic, 7th April 2019*, volume 291 of *EPTCS*, pages 24–34, 2019. `doi:10.4204/EPTCS.291.3`.

18 Elena Giachino, Cosimo Laneve, and Michael Lienhardt. A framework for deadlock detection in core ABS. *Software and Systems Modeling*, 15(4):1013–1048, 2016. `doi:10.1007/s10270-014-0444-y`.

19 Dilian Gurov, Reiner Hähnle, and Eduard Kamburjan. Who carries the burden of modularity? In Tiziana Margaria and Bernhard Steffen, editors, *Leveraging Applications of Formal Methods, Verification and Validation, 9th Intl. Symp., ISoLA 2020, Rhodes, Greece*, LNCS. Springer, October 2020.

20 Anton W Haubner. Semi-dynamic session types for ABS. Bachelor thesis, Technical University of Darmstadt, 2019. URL: `https://github.com/ahbnr/SessionTypeABS/blob/master/thesis/thesis_final_pdfa.pdf`.

21 Carl Hewitt, Peter Bishop, and Richard Steiger. A universal modular ACTOR formalism for artificial intelligence. In *Proceedings of the 3rd International Joint Conference on Artificial Intelligence*, IJCAI'73, pages 235–245. Morgan Kaufmann Publishers Inc., 1973. URL: `http://dl.acm.org/citation.cfm?id=1624775.1624804`.

22 Kohei Honda, Nobuko Yoshida, and Marco Carbone. Multiparty asynchronous session types. In George C. Necula and Philip Wadler, editors, *Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2008, San Francisco, California, USA, January 7-12, 2008*, pages 273–284. ACM, 2008. `doi:10.1145/1328438.1328472`.

23 Atsushi Igarashi, Peter Thiemann, Vasco T. Vasconcelos, and Philip Wadler. Gradual session types. *Proc. ACM Program. Lang.*, 1(ICFP):38:1–38:28, 2017. `doi:10.1145/3110282`.

24 Einar Broch Johnsen, Reiner Hähnle, Jan Schäfer, Rudolf Schlatte, and Martin Steffen. ABS: A core language for abstract behavioral specification. In Bernhard K. Aichernig, Frank S. de Boer, and Marcello M. Bonsangue, editors, *FMCO 2010*, volume 6957 of *Lecture Notes in Computer Science*, pages 142–164. Springer, 2010. `doi:10.1007/978-3-642-25271-6_8`.

25 Eduard Kamburjan. Detecting deadlocks in formal system models with condition synchronization. *ECEASST*, 76, 2018. `doi:10.14279/tuj.eceasst.76.1070`.

26 Eduard Kamburjan. Behavioral program logic. In *TABLEAUX*, volume 11714 of *Lecture Notes in Computer Science*, pages 391–408. Springer, 2019.

27 Eduard Kamburjan. *Modular Verification of a Modular Specification: Behavioral Types as Program Logics*. PhD thesis, Technische Universität Darmstadt, 2020.

28 Eduard Kamburjan and Tzu-Chun Chen. Stateful behavioral types for active objects. In Carlo A. Furia and Kirsten Winter, editors, *iFM 2018*, volume 11023 of *Lecture Notes in Computer Science*, pages 214–235. Springer, 2018. `doi:10.1007/978-3-319-98938-9_13`.

29 Eduard Kamburjan, Crystal Chang Din, and Tzu-Chun Chen. Session-based compositional analysis for actor-based languages using futures. In Kazuhiro Ogata, Mark Lawford, and Shaoying Liu, editors, *ICFEM 2016*, volume 10009 of *Lecture Notes in Computer Science*, pages 296–312, 2016. `doi:10.1007/978-3-319-47846-3_19`.

30 Eduard Kamburjan, Crystal Chang Din, Reiner Hähnle, and Einar Broch Johnsen. Asynchronous cooperative contracts for cooperative scheduling. In *SEFM*, volume 11724 of *Lecture Notes in Computer Science*, pages 48–66. Springer, 2019.

31 Eduard Kamburjan, Reiner Hähnle, and Sebastian Schön. Formal modeling and analysis of railway operations with active objects. *Sci. Comput. Program.*, 166:167–193, 2018. `doi:10.1016/j.scico.2018.07.001`.

32 Eduard Kamburjan, Stefan Mitsch, Martina Kettenbach, and Reiner Hähnle. Modeling and verifying cyber-physical systems with hybrid active objects. *CoRR*, abs/1906.05704, 2019. `arXiv:1906.05704`.

33 Michael Kaminski and Nissim Francez. Finite-memory automata. *Theor. Comput. Sci.*, 134(2):329–363, 1994. `doi:10.1016/0304-3975(94)90242-9`.

34 Rumyana Neykova, Laura Bocchi, and Nobuko Yoshida. Timed runtime monitoring for multiparty conversations. *Formal Asp. Comput.*, 29(5):877–910, 2017.

35 Rumyana Neykova and Nobuko Yoshida. Multiparty session actors. *Logical Methods in Computer Science*, 13(1), 2017.

36 Mila Dalla Preda, Maurizio Gabbrielli, Saverio Giallorenzo, Ivan Lanese, and Jacopo Mauro. Dynamic choreographies - safe runtime updates of distributed applications. In Tom Holvoet and Mirko Viroli, editors, *COORDINATION 2015*, volume 9037 of *Lecture Notes in Computer Science*, pages 67–82. Springer, 2015. `doi:10.1007/978-3-319-19282-6_5`.

37 Mila Dalla Preda, Maurizio Gabbrielli, Saverio Giallorenzo, Ivan Lanese, and Jacopo Mauro. Dynamic choreographies: Theory and implementation. *Log. Methods Comput. Sci.*, 13(2), 2017. `doi:10.23638/LMCS-13(2:1)2017`.

38 Alceste Scalas and Nobuko Yoshida. Lightweight session programming in scala. In Shriram Krishnamurthi and Benjamin S. Lerner, editors, *30th European Conference on Object-Oriented Programming, ECOOP 2016, July 18-22, 2016, Rome, Italy*, volume 56 of *LIPIcs*, pages 21:1–21:28. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2016. `doi:10.4230/LIPIcs.ECOOP.2016.21`.

39 Rudolf Schlatte and abstools Contributors. Modified branch of the abstools compiler version 1.8.1 - github source repository. `https://github.com/ahbnr/abstools/tree/thisDestiny`. Accessed: 2019-10-29.

40 Rudolf Schlatte, Einar Broch Johnsen, Jacopo Mauro, Silvia Lizeth Tapia Tarifa, and Ingrid Chieh Yu. Release the beasts: When formal methods meet real world data. In Frank S. de Boer, Marcello M. Bonsangue, and Jan Rutten, editors, *It's All About Coordination - Essays to Celebrate the Lifelong Scientific Achievements of Farhad Arbab*, volume 10865 of *Lecture Notes in Computer Science*, pages 107–121. Springer, 2018. `doi:10.1007/978-3-319-90089-6_8`.

41 Robert E. Strom and Shaula Yemini. Typestate: A programming language concept for enhancing software reliability. *IEEE Trans. Software Eng.*, 12(1):157–171, 1986. `doi:10.1109/TSE.1986.6312929`.

42 Roger Wolff, Ronald Garcia, Éric Tanter, and Jonathan Aldrich. Gradual typestate. In Mira Mezini, editor, *ECOOP 2011 - Object-Oriented Programming - 25th European Conference, Lancaster, UK, July 25-29, 2011 Proceedings*, volume 6813 of *Lecture Notes in Computer Science*, pages 459–483. Springer, 2011. `doi:10.1007/978-3-642-22655-7_22`.

43 Peter Y. H. Wong, Elvira Albert, Radu Muschevici, José Proença, Jan Schäfer, and Rudolf Schlatte. The ABS tool suite: modelling, executing and analysing distributed adaptable object-oriented systems. *STTT*, 14(5):567–588, 2012.